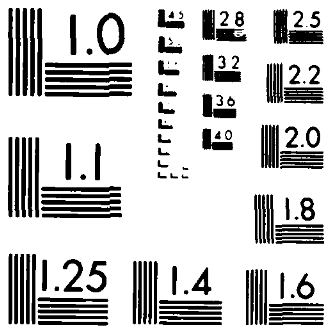
AD-A149 233 NAMING IN PROGRAMMING(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA N C ROWE OCT 84 NPS52-84-017 1/1

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

(2)

NPS52-84-017

NAVAL POSTGRADUATE SCHOOL

Monterey, California

NAMING IN PROGRAMMING

Neil C. Rowe

October 1984

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Commodore R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

Neil C. Rowe

NEIL C. ROWE
Associate Professor of
Computer Science

Reviewed by:

Bruce MacLennan

BRUCE J. MAC LENNAN
Department of Computer Science

Released by:

KNEALE T. MARSHALL
Dean of Information and
Policy Sciences

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-84-017	2. GOVT ACCESSION NO. AD-A149 233	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Naming in programming		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Neil C. Rowe		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; RR000-01-10 N0001484WR41001
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE October 1984
		13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

programming languages, identifiers, names, education -- computer science, structured programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Good names are important in programming. Names do not affect program performance, but can make program text clearer. In particular, they can bring out family relationships between similar procedures. I discuss common obstacles to good naming. I then give some examples of good naming in the language Logo, a language in which naming (with the TO construct) is a central metaphor for the programmer's activity. The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds (continued)

DD FORM 1473, 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Continuation of block 20

provided by the Chief of Naval Research.

This paper is to appear in Computers in the Schools.

A-1

Naming in Programming

Neil C. Rowe, Ph.D.

Department of Computer Science
Code 52
Naval Postgraduate School
Monterey, California 93943

ABSTRACT ./. Author

Good names are important in programming. Names do not affect program performance, but can make program text clearer. In particular, they can bring out family relationships between similar procedures. I discuss common obstacles to good naming. I then give some examples of good naming in the language Logo, a language in which naming (with the TO construct) is a central metaphor for the programmer's activity.

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research. This paper is to appear in *Computers in the Schools*.

1. INTRODUCTION

Names are an essential but underrated aspect of programming. Many programming and programming language issues can be addressed in the study of naming.

By "names" I mean the identifiers (character strings) used to distinguish (a) program units (procedures, subroutines, blocks, loops, etc.) and (b) variables. I shall particularly emphasize procedure names in this discussion, since they are usually the most important. The choosing of names for a given program, what might be called "name programming", is an issue of program semantics as opposed to syntax. Program comments are also a kind of semantics, but a less satisfactory one: a name's denotations and connotations are tightly bound to that name, and are invoked every time that name is used.

I dispute the common view that "names are arbitrary and a matter of taste" or "it doesn't really matter what you call things, so long as they work." Just as with the English language, there are always options for names, but these options are small compared to the enormous number of limiting constraints imposed by syntactic and semantic conventions. To the extent that programming is a social activity involving more than one programmer, conventional names must be agreed on -- and the closer names are to English, with occasional exceptions, the easier will be comprehension by all concerned. And names can be "right" or "wrong": take for instance a programmer who labels a square root procedure SQUARE and a square procedure SQUAREROOT -- the names are not so much "inappropriate" as wrong.

1.1. Children and Naming

It's interesting to observe the naming problems of children when programming. Here are a teacher's notes about sixth graders using Logo, plus some extra software, to make drawings (from Goldenberg, 1976):

> "Item: John finishes drawing a picture and wants to 'teach' the turtle how to do it by itself. The computer types WHAT IS THE NAME OF THE PROCEDURE? and John sighs and says, "Oh, no, not another name!' He says it with amusement, but is clearly troubled by the difficulty of finding a name. Robert, too, often says, 'What should I name that?' The 'should' instead of 'what' is significant. Sally says, 'I don't know what to call that?'"
>
> "Item: Robert began picking up lots of local words for naming his procedures. His MAC truck may have been influenced somewhat by having heard of Mack Trucks, but Robert got the name directly from our Project MAC sign. Many kids immortalize siblings and classmates by naming procedures after them..."
>
> "Item: Theme names! Robert was creating an animation of a truck that drives into a wall and crumples and then keeps backing up and battering the wall over and over again. The original truck name was MAC. PUTTPUTT, CRASH, and SMASH were also parts of the program, but one cannot tell by the names alone what each part does. The fact that these names were still really somewhat arbitrary for Robert can be seem from the other names of procedures in the overall program: OZ, CA, COB, TV, SCRIB, JO, HOJO, and CAT. Sally used THINKER, ROBOT, EARLESS, BODY, BODYALL, ARMLESS and SLOWPOKE for her robot, but again, even the ones like ARMLESS, EARLESS, BODY, and BODYALL, which suggest stages of development, were not consistently used that way."
> [pages 1-2]

Adults are just grown up children, and it's not so much that these problems don't arise with adults as it is that adults conceal problems better. Too long procedures may be evidence for a naming hangup, as may be strongly numeric (e.g. P63RS007) and allusive names (e.g. SHRDLU).

1.2. Reason 1 for Poor Naming: Existing Language Limitations

Unfortunately, some current computer languages have length limitations on names. This is more tolerable and widespread at the systems programming level, but some programming languages are still afflicted with this malady. Length limitations on names save a mere one extra pointer in storage, which is not worth the sacrifice in clarity.

1.3. Reason 2 for Poor Naming: Lack of Naming Aids

Even when the length of names is not limited in a language, the lack of positive aids to the use of them may be seriously constraining. Typing of names is a time-consuming part of programming. If longer names are to be used in programming, the amount of typing will be increased unless abbreviations and other aids are used. We will discuss such aids later in this paper.

1.4. Reason 3 for Poor Naming: Legacy of Mathematics

Programming's origins are in mathematics, where one-letter variable names are the norm. Most computer applications today have nothing to do with mathematics, so we no longer need respect this convention. Besides, the reason for one-letter names in mathematics seems to be the desire to make multiplication a "default" operation (so ABC means A times B times C). This makes sense for math since multiplication is common. But for most programming, "word construction" makes a better default operation.

1.5. Reason 4 for Poor Naming: Ideas That Are Hard to Articulate

The most basic reason for poor naming may be that certain ideas are difficult to put in words. (Rowe, 1980) discusses this in regard to the teaching of grammar, but another good example is music. Suppose one represents a musical composition as a list of triples (containing pitch frequency, note starting time, and note duration), and suppose there are a thousand such triples (notes) in the composition. We can usually figure out what the main parts of the composition are and divide it into procedural parts. But what do we name those parts? What music seems to be "about" is hard to put into words. This seems to be true of things that are time-critical: people have a hard time analyzing things that happen once and are then over.

But assigning names is part of what science is about. Being forced to think up a name is often a good thing. For instance, in teaching Logo programming I have required student programs to be no more than five lines long. This is occasionally awkward, but it does seem to encourage the development of naming skills as well as the use of structured programming. So it could be that the difficulties one has in finding words to describe musical thoughts is more a reflection on the primitive stage of musical analysis (which is, after all, not a high priority research area) than of any inherent naming limitations.

2. SIMPLE DESCRIPTIVE NAMES

What is good naming? Clearly it is using names that are evocative, and which conjure up distinctive characteristics of what is being named. For procedure names this usually means finding some English word that represents the primary accomplishment of the procedure. For instance, a procedure that draws a triangle could be called TRIANGLE, a procedure that finds item number n in a list could be called ITEM, and a procedure that generates and prints out sentences could be called SENTENCES. When naming variables, a variable that is a number could be called NUMBER, and a variable that is a list that represents a sentence could be called SENTENCE. Names of more than one word can be made with punctuation characters like the period in Logo. For example, a Logo procedure that draws a big triangle could be BIG.TRIANGLE, and a Logo procedure that generates random sentences could be called RANDOM.SENTENCES.

Certain names may be better in certain languages. For example, there is a bias towards verbs as procedure names in Logo. This is probably because of the use of the word TO.

But description is not the whole story. A good name should not merely characterize something, but should also be the most general characterization and no more. For instance, consider the following program in Logo (Abelson, 1982). This classic program draws a triangle of a given size. To run it, the user types the word TRIANGLE, a space, a number representing the desired size, and a carriage return.

```
TO TRIANGLE :SIDE
FORWARD :SIDE
RIGHT 120
FORWARD :SIDE
RIGHT 120
FORWARD :SIDE
RIGHT 120
END
```

Young students using Logo will tend to name this procedure ROOF or give it the name of some other concrete object. While this does capture a valuable use of such a procedure, it does not characterize the multitude of other legitimate uses (as a mountain, say). A better name (for adult programmers, at least) would be an abstract, geometrical one like TRIANGLE. After all, the shape drawn on the video screen is merely three thin lines; there are no shingles on this ROOF. On the other hand, we don't want to get too general: the name SHAPE would describe this procedure, but would apply to squares and other shapes. So good naming involves choosing just the right level of generality. The approximate level does, however, vary with the sophistication of the programmer and the likelihood he will reuse it for a different purpose. ROOF would probably be a good name for nine-year-olds, but not for college students.

Good naming is important in other languages such as Smalltalk (Goldberg and Robson, 1983). The issues concerning naming are similar for Smalltalk except that more system-defined names are already provided.

3. COMPOSITE NAMES AND NAME FAMILIES

From the preceding, we see that good naming involves aesthetic issues. A good name evokes a distinctive picture in a person's head, and one can can be an art critic. But there is another, more formulable aspect of names: their characterization of relationships and "family resemblances" between procedures. In this section we will investigate this.

3.1. The ITEM Family

Consider the following Logo procedure (in the Apple Logo dialect, as will be the rest of our examples):

```
TO ITEM :N :L
TEST :N = 1
IFTRUE OUTPUT FIRST :L
IFFALSE OUTPUT ITEM (:N - 1) (BUTFIRST :L)
END
```

Read the first line as "Get item number N of list L", interpreting the N and the L to be names of stereotypical integer and list parameters, respectively. So if you type PRINT ITEM 3 [TOM DICK HARRY JOE] it would print "HARRY". The use of the one-letter variable names N and L could be a little confusing -- perhaps we should have called them NUMBER and LIST -- but has the advantage of distinguishing them well from procedure names. Long variable names are better for special-purpose

variables, which require a good self-description.

ITEM is one of a whole "family" of related procedures:

```
TO RUN.ITEM :N :L
TEST :N = 1
IFTRUE RUN FIRST :L
IFFALSE RUN.ITEM (:N - 1) (BUTFIRST :L)
END

TO FIRST.N.ITEMS :N :L
TEST :N = 1
IFTRUE OUTPUT (LIST FIRST :L)
IFFALSE OUTPUT SENTENCE (FIRST :L)
   (FIRST.N.ITEMS (:N - 1) (BUTFIRST :L))
END

TO RANDOM.ITEM :L
TEST 0 = RANDOM LENGTH :L
IFTRUE OUTPUT FIRST :L
IFFALSE OUTPUT RANDOM.ITEM BUTFIRST :L
END
```

These procedure names can be read as:

"RUN item number N of list L."
"Get first N items of list L."
"Get a random item from list L."

For example:

RUN.ITEM 3 [[FORWARD 10] [FORWARD 25] [RIGHT 90] [RIGHT 180]]
turns the turtle right 90 degrees,

PRINT FIRST.N.ITEMS 3 [TOM DICK HARRY JOE BILL]
prints the list [TOM DICK HARRY], and

PRINT RANDOM.ITEM [1 2 3 4 5 6 7 8 9]
prints a random digit.

The LENGTH procedure mentioned in RANDOM.ITEM is a recursive procedure that counts the length of a list L, defined as follows:

```
TO LENGTH :L
IF :L = [] THEN OUTPUT 0 ELSE OUTPUT (1 + LENGTH BUTFIRST :L)
END
```

Note the basic structural similarity of all these procedures to ITEM. This is mirrored by the "adjective-noun" form of the names, where the noun is ITEM or ITEMS and the adjective describes what kind of an ITEM variant the procedure is. Here is an inverse ITEM:

```
TO INVERSE.ITEM :I :L
TEST :I = FIRST :L
IFTRUE OUTPUT 1
IFFALSE OUTPUT (1 + INVERSE.ITEM :I (BUTFIRST :L))
END
```

The first line can be read as "Do the inverse of the situation where the number N item of list L is I." So PRINT INVERSE.ITEM 10 [5 10 25 50 100] prints the number 2, since 10 is the second item of the list. Note the similarity in form between INVERSE.ITEM and ITEM.

3.2. The PROPERTY Family

Consider now a analogous family of procedures operating on a list of pairs (a "property list") rather than a regular list:

```
TO PROPERTY :I :PL
TEST :I = FIRST FIRST :PL
IFTRUE OUTPUT LAST FIRST :PL
IFFALSE OUTPUT PROPERTY :I (BUTFIRST :PL)
END

TO RUN.PROPERTY :I :PL
TEST :I = FIRST FIRST :PL
IFTRUE RUN LAST FIRST :PL
IFFALSE RUN.PROPERTY :I (BUTFIRST :PL)
END

TO FIRST.N.PROPERTIES :N :PL
TEST :N = 1
IFTRUE OUTPUT (LIST LAST FIRST :PL)
IFFALSE OUTPUT SENTENCE (LAST FIRST :PL)
  (FIRST.N.PROPERTIES (:N - 1) (BUTFIRST :PL))
END

TO RANDOM.PROPERTY :PL
TEST 0 = RANDOM LENGTH :PL
IFTRUE OUTPUT LAST FIRST :PL
IFFALSE OUTPUT RANDOM.PROPERTY BUTFIRST :PL
END
```

Here I represents an "item" variable, and PL a pair list. Note we can also have an inverse:

```
TO INVERSE.PROPERTY :I :PL
TEST :I = LAST FIRST :PL
IFTRUE OUTPUT FIRST FIRST :PL
IFFALSE OUTPUT INVERSE.PROPERTY :I (BUTFIRST :PL)
END
```

The names of these procedures specify precise analogies to the ITEM family procedures. We can read the five procedure names as follows:

"Get property I of property (pair) list PL."
"RUN property I of property list PL."
"Get the values of the first N properties in property list PL."
"Get a random property value from property list PL."
"Get the property name in property list PL whose value is I."

Some examples:

PRINT PROPERTY 3 [[SUNNY .5] [CLOUDY .3] [PARTLYCLOUDY .2] [RAIN .1]]
prints the number .2

RUN.PROPERTY 2 [[SUNNY [CLEARSCREEN PRINT [GET OUTDOORS]]] [RAIN

```
PRINT [NICE DAY TO BE INSIDE]]]]
prints [NICE DAY TO BE INSIDE]

FIRST.N.PROPERTIES 2 [[SUNNY .5] [CLOUDY .3] [PARTLYCLOUDY .2] [RAIN .1]]
prints [.5 .3]

RANDOM.PROPERTY [[SUNNY .5] [CLOUDY .3] [PARTLYCLOUDY .2] [RAIN .1]]
gives a random number from .5, .3, .2, and .1

INVERSE.PROPERTY .3 [[SUNNY .5] [CLOUDY .3] [PARTLYCLOUDY .2] [RAIN .1]]
prints CLOUDY
```

3.3. The RANDOM Family

The above two families can be thought of as "parallel" since for each procedure in one there is a corresponding procedure in the other. Families can also be "orthogonal" to one another, "intersecting" at a single procedure. For instance, the RANDOM family of which we have seen the members RANDOM.ITEM and RANDOM.PROPERTY also includes a procedure to choose a random bunch of items from a list:

```
TO RANDOM.SUBSET :L
IF :L = [] THEN OUTPUT []
TEST 0 = RANDOM 1
IFTRUE OUTPUT SENTENCE (LIST FIRST :L) (RANDOM.SUBSET BUTFIRST :L)
IFFALSE OUTPUT RANDOM.SUBSET BUTFIRST :L
END
```

It also includes more numerical procedures, like this random number generator that tends to return numbers at the middle of its range more often than other numbers:

```
TO BIASED.RANDOM :N
OUTPUT QUOTIENT ((RANDOM :N) + (RANDOM :N)) 2
END
```

And it includes more basic procedures as well, like the built-in function RANDOM. Note that the names of procedures in these families, having the "adjectives" in common between their names, have less syntactic similarity as opposed to underlying "semantic" similarity. For the RANDOM family this underlying semantic similarity is the notion of pseudo-random choice among options.

Just as RANDOM.PROPERTY relates to PROPERTY, RANDOM.SUBSET relates to a procedure SUBSET, which gives the subset of items in a set having some particular value (which could be useful for counting those values):

```
TO SUBSET :I :L
IF :L = [] THEN OUTPUT []
TEST :I = FIRST :L
IFTRUE OUTPUT SENTENCE (FIRST :L) (SUBSET :I (BUTFIRST :L))
IFFALSE OUTPUT SUBSET :I (BUTFIRST :L)
END
```

So this and RANDOM.SUBSET form a family, which we might call the SUBSET family. But its analogous intersections with the RUN, FIRST.N, and INVERSE families do not seem to be meaningful.

3.4. The Test-Operator Family

RANDOM.ITEM relates to ITEM in the same way that RANDOM itself (the procedure that generates random numbers) might relate to a procedure IDENTITY that merely outputs its single argument. (Unfortunately we cannot give procedures names that are zero characters long, as the analogy between the names might suggest, so we compromise on "IDENTITY".) RANDOM.ITEM relates to RUN.ITEM in the same way that RANDOM relates to RUN. So we can create a new family including basic "test functions" like IDENTITY, RUN, and RANDOM, whose names are prefixes stripped off the corresponding members of the ITEM or PROPERTY families. Unfortunately there is no counterpart in this family for FIRST.N.ITEMS and INVERSE.ITEM.

3.5. Diagram of the Families

Figure 1 shows the families of procedures we have mentioned so far. (Intersection points not marked represent nonmeaningful combinations.)

3.6. Version Families

There is also a "third dimension" for which we can build families. In the third dimension are procedures that accomplish the same thing in different ways. These procedures can be denoted by suffixes on the procedure name. For instance, we've assumed a "default" of recursive procedures; we could also write ITEM iteratively, and call it ITEM.ITERATIVE. We could also have a ITEM.FLOATING that took a floating-point number for N and rounded it to the nearest integer to get an index into a list. Or we could just have different "versions" (perhaps with bugs) of the same general structure:

```
TO ITEM.3 :N :L
TEST :N > 1
IFTRUE OUTPUT FIRST :L
IFFALSE OUTPUT ITEM.3 (:N - 1) (BUTFIRST :L)
END
```

The version identifier can be put at the end of the name, after the other suffixes, as is done with file names in Interlisp.

3.7. Hierarchies Vs. Families

This infinite multi-dimensional structure is independent of the ways procedures are related hierarchically in programs. Rather, it is an abstraction showing where names originate, a Platonic supermarket of ideal procedures. In fact, it is a good idea for names of procedures in programs to be mostly unrelated, to bring out differences in functions. For example, we could have written RANDOM.ITEM non-recursively by a call to ITEM, using the LENGTH procedure defined above:

```
TO RANDOM.ITEM.COMPOSED :L
OUTPUT ITEM (1 + (RANDOM LENGTH :L)) :L
END
```

But this is different in form from our previous RANDOM.ITEM, and demands a name that calls attention to this, as for instance RAND.ITEM.

3.8. Multiple Names

Names need not be single. Under different circumstances, a procedure may be performing a different function, and deserve a different name. An example would be the use of TRIANGLE for both roofs and mountains. Or a specific purpose in a program may be served by a quite general procedure, and a more specific name can explain what's going on. But this should not be overdone or it may obscure underlying similarities of procedure usage.

4. RECOMMENDATIONS FOR PROGRAMMING LANGUAGES

Our above discussion suggests that programming languages, especially those used by students, should provide more support for longer, more descriptive names. I offer a few suggestions.

4.1. Spelling Correction

The probability of a spelling error is approximately proportional to the length of words used. Hence automatic spelling correction is essential to support programming with long names. This is not as hard as is often believed; several good recent algorithms can make it quite fast, and microcoding allows further speedup. Interlisp, for instance, has automatic spelling correction, but most versions of Logo do not.

4.2. Abbreviation Expansion

Going a step further, we can apply the same ideas to abbreviations, words purposely "misspelled" in a way so as to make their lengths considerably shorter. Abbreviations tend not to be just random deletions of letters: letters on the end go first, then vowels go before consonants, and so on. So there are rules of abbreviation that could be exploited to automatically expand abbreviations typed by a user.

4.3. Consistency Connections Between Names

With long names, it can be hard work to change a subname in every procedure in which it occurs. Simple methods (like pointers in the symbol table) could link together all occurrences.

4.4. Diagnostics From Names

Since names are a rich source of information about what a program is trying to do, they should be exploited in debugging. This is hard in general, but is possible if a user supplies information to a compiler or interpreter about their conventions (or if the user agrees to follow certain conventions). For instance, nonanalogous structure for one member of a family of procedures could be noted by a warning diagnostic.

5. CONCLUSIONS

I have put forth some ideas about naming in programming, mostly relating to the naming of procedures and how to make good naming easier. I discussed some of the serious obstacles in programming languages to good naming. Good naming was then illustrated with an example of some "families" of procedures that interrelate, and how their names can reflect their relationships. I then suggested a few ways to improve programming languages to support this kind of good naming. These comments apply to any programming language, but are particularly directed towards Logo since naming is emphasized in most pedagogical aids to the language.

6. REFERENCES

Abelson, H. (1982). *Logo for the Apple II.* Peterborough NH: BYTE/McGraw-Hill, 1982.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation.* Reading MA: Addison-Wesley, 1983.

Goldenberg, E. P. (1976). Children's strategies for naming procedures. Unpublished Logo Working Paper #58, MIT Logo Project, Cambridge MA.

Rowe, N. C. (1980). Inductive common ground tutoring. *Computers and Education, 4,* 2, 177-187.

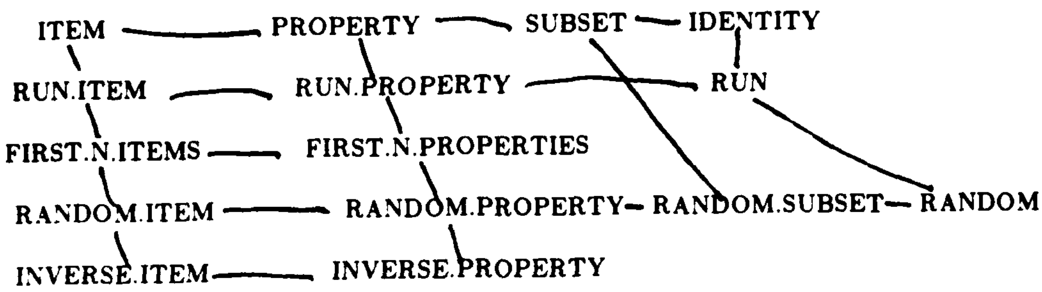

ITEM
PROPERTY
SUBSET
IDENTITY
RUN.ITEM
RUN.PROPERTY
RUN
FIRST.N.ITEMS
FIRST.N.PROPERTIES
RANDOM.ITEM
RANDOM.PROPERTY
RANDOM.SUBSET
RANDOM
INVERSE.ITEM
INVERSE.PROPERTY

Figure 1

DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	3
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93943	40
Associate Professor Neil C. Rowe, Code 52Rp Department of Computer Science Naval Postgraduate School Monterey, CA 93943	25
Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217	1
Dr. David W. Mizell Office of Naval Research 1030 East Green Street Pasadena, CA 91106	1

END

FILMED

2-85

DTIC